VOL. 5 NO. 1 JAN. 1988

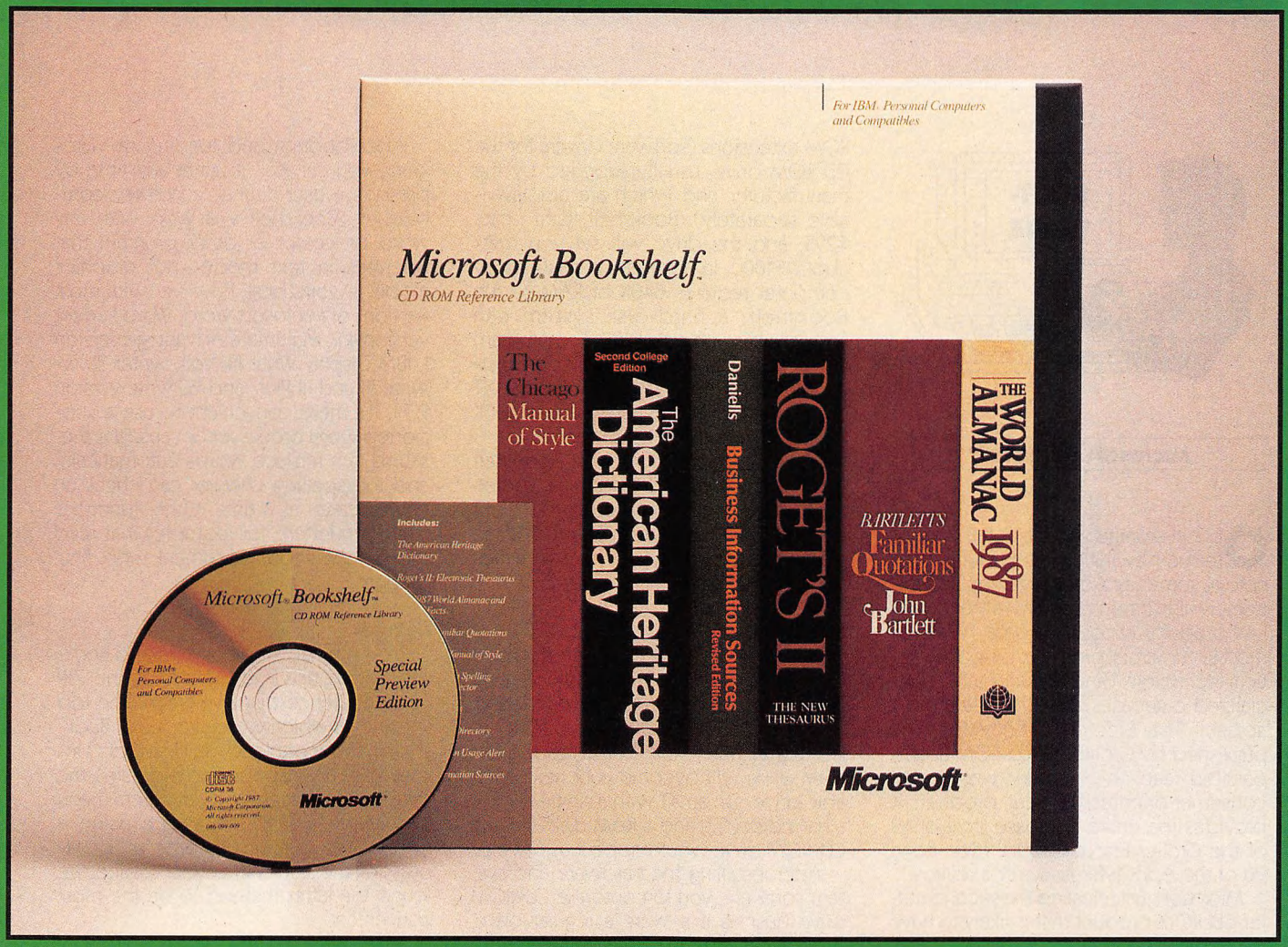
COMPUTERDIGEST

A NEW KIND OF MAGAZINE FOR ELECTRONICS PROFESSIONALS

MICROSOFT BOOKSHELF

CD ROM Reference Library

Page 88



FLOPPY DISKS

Bits, bytes, and copy protection

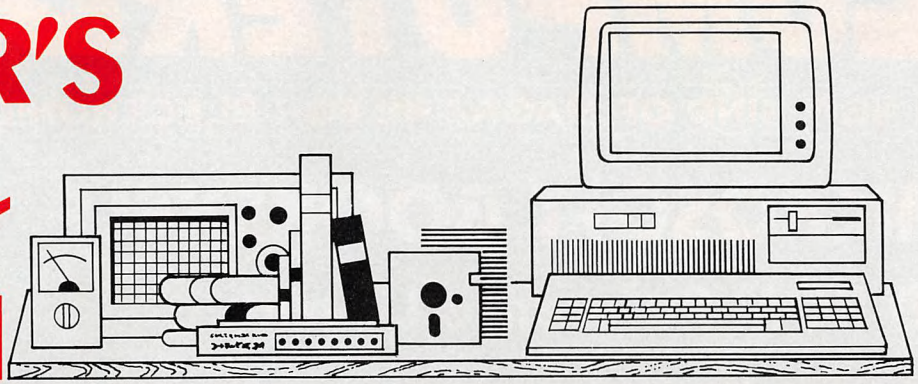
Page 90

68000 COMPUTING

The CD Classroom continues

Page 96

EDITOR'S WORK-BENCH



Microsoft Bookshelf

Optical storage will be at the core of the next evolution in computer technology, some say. Advantages include extremely high information density and very low production costs. To date, however, most commercial products based on optical technology (financial and other specialized databases) have had little mass appeal. The Electronic Encyclopedia published by Grolier on CD-ROM (Compact Disk read-only memory) was the first consumer-oriented optical product; it provides the entire database (text only) of the Grolier Encyclopedia. (See page 90 of the April 1987 issue for a review.)

Microsoft just released the second major CD-ROM product that is likely to have widespread appeal; it is called Bookshelf. Bookshelf consists not of a single database, like the Electronic Encyclopedia, but of ten reference works, including: *The American Heritage Dictionary, Roget's II Thesaurus, The World Almanac and Book of Facts, Bartlett's Familiar Quotations, The Chicago Manual of Style, The U.S. ZIP Code Directory,* a spelling checker, a usage checker, a set of form letters, and a categorized listing of sources of business information.

To run Bookshelf you need a CD-ROM drive, a PC, XT, AT, or clone, DOS 3.0 or higher, and a copy of the MS-DOS CD-ROM extensions (software drivers for the CD-ROM drive, usually supplied by the manufacturer, and which are also available separately). Bookshelf itself costs $295, and the drive will set you back about $1000. In addition, a dual-floppy computer requires 640K of RAM to run Bookshelf; a hard-disk system can squeeze by with only 512K. The program will run on a dual-floppy system. We tested Bookshelf on a standard IBM PC XT with a 20-megabyte hard disk, a 500K RAM disk, and a 500K disk cache.

At those prices, obviously, Bookshelf will not compete with paperback reference works sold at dime stores, but that's not Bookshelf's intended market. The target, rather, is the serious writer—including the serious technical writer.

Setting up

The CD-ROM drive itself interfaces to your PC via a cable that connects to a plug-in expansion card. After installation, which includes modifying your CONFIG.SYS and AUTOEXEC.BAT files, the CD-ROM drive will become your next available drive. On an XT with a single floppy, a hard disk (C:), and a RAM disk (D:), the CD-ROM drive would appear as drive E.

After installing the hardware and system software, you log onto the CD-ROM drive (just as if it were a regular drive, except that you can't write to it), and run a Setup utility. That program copies files to your hard disk and installs the program according to the hardware and software you'll be using Bookshelf with.

Normally you run Bookshelf in a memory-resident mode so that you can call it up from within your word processor while you are writing. (You can also run Bookshelf as a transient application.) Bookshelf requires about 128K of memory (in memory-resident mode), and really works better with a RAM disk, which you'll probably want to locate in the EMS memory.

After booting and loading the program, you can call it up at any time by pressing a user-definable hot-key combination. Bookshelf will work with any word processor or other program that operates in text mode—not graphics mode. It works best, however, with most versions of Microsoft Word, Word Perfect version 4.2, Multimate Advantage version I, IBM Display Write III, Volkswriter 3, XyWrite III and III Plus, and PC Write version 2.71. It's more convenient to use a supported word processor because the thesaurus can replace words automatically, and the spelling checker can check an entire screen at a time. Other functions (the form letters, for example) that feed information into a document work fine, though.

Using it

Press your defined hot keys, and Bookshelf pops up. Figure 1 shows the ZIP Code reference display screen. The top line shows which word processor Bookshelf has recognized (it displays "Unknown" in an unsupported program); the second line presents a twelve-item menu bar which you can navigate using a mouse, the cursor-control keys, or by pressing the first letter of your choice. We found the latter method to be the most convenient.

After selecting the desired reference work, a menu appears, allowing various options for the chosen work. The dictionary's menu, for example, allows to look up a word, biographical information, or geographical information.

Next a dialog box appears, in which you enter additional information. Dialog boxes work differently according to which opus you're working with. The ZIP Code checker, for example, will pick an address off the screen and find the correct ZIP Code. Bookshelf will then feed the ZIP Code back into your word processor just as if you had typed it. Alter-

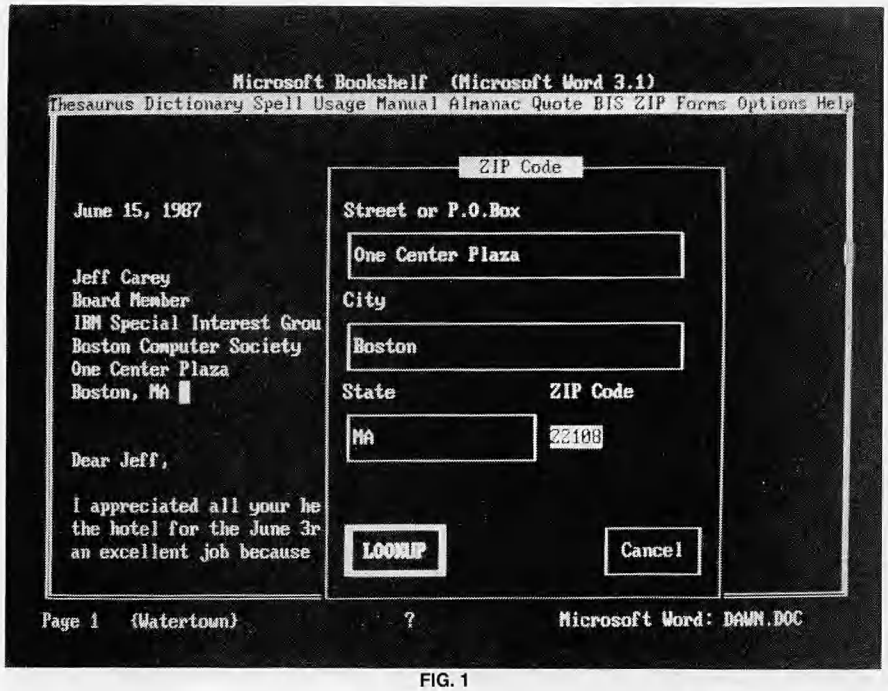


FIG. 1

natively, you can type an address in the dialog box, look up the ZIP Code, and feed the whole thing into your word processor.

The ability to look up ZIP Codes is useful, if unexciting. Bookshelf's real strengths are the dictionary, the thesaurus, the book of quotations, and the almanac. The dictionary is based on the full American Heritage dictionary; it includes complete definitions (not simple "synonyms"), word derivations, and biographical and geographical information. You can look up a word that Bookshelf reads from the screen or you can type one in from a dialog box. In addition, you can do cross-index searches—find all entries in which the word *computer* is used, for example. Unfortunately, you cannot paste definitions into your documents (presumably for copyright reasons). You can scroll through an alphabetized list of entry words only, without displaying definitions, and zoom in on one by pressing a key. In the expanded display, highlighting is used to indicate word entries, syllable breaks, derived forms, the definitions themselves, and derivation. Synonyms and points of usage are discussed at some entries, just as in the "real" (paper) version of the dictionary. For browsing, you can "open" the dictionary at the beginning of any letter—*A* for example.

Accessing the other works functions in a similar manner. However, locating information can be quirky, if not difficult. For example, I wanted to locate financial information in the almanac about the computer industry. After entering the words *computer* and *finance* in the dialog box, I was presented with a list of articles. I selected one, and then read it. I wanted to return to the list of articles to choose another, but was unable to do so without re-entering the search words and waiting for Bookshelf to search its database. In general, depending on the type of search, you may end up doing a lot of thumb twiddling.

Conclusions

Bookshelf is a wonderful idea, but the implementation leaves something to be desired. In spite of the fact that it uses 128K of RAM and can use a RAM disk for temporary storage, it is slow. Search speed is slow, and display speed, once the information is found, is slow. In fact, text spills across the screen at a rate that is comparable to what a 1200-baud modem provides.

Bookshelf's strong point is the dictionary, which is truly a dictionary, unlike all other products for personal computers. The thesaurus and the quotations reference also work well. The almanac contains a great deal of information, but getting at it is a problem, as described in detail above.

As a professional editor and writer, I want to like Bookshelf, and I want the capabilities it can provide. But the program's slow performance, coupled with the high cost of the product and the hardware necessary to run it, cause me to question whether I couldn't get along without it. Admittedly, Bookshelf can do some things that would be impossible by hand—doing comprehensive searches, for example, or directly importing text into a document—but bread-and-butter searching and browsing can usually be done much more efficiently the old-fashioned way.

Microsoft is to be commended for bringing Bookshelf to market; eventually all personal computers will have comparable capabilities. Let's hope that the company sees fit to upgrade Bookshelf's performance to what it should be, and that CD-ROM drives fall in price to an affordable level.

Borland-Osborne/McGraw-Hill Programming Series

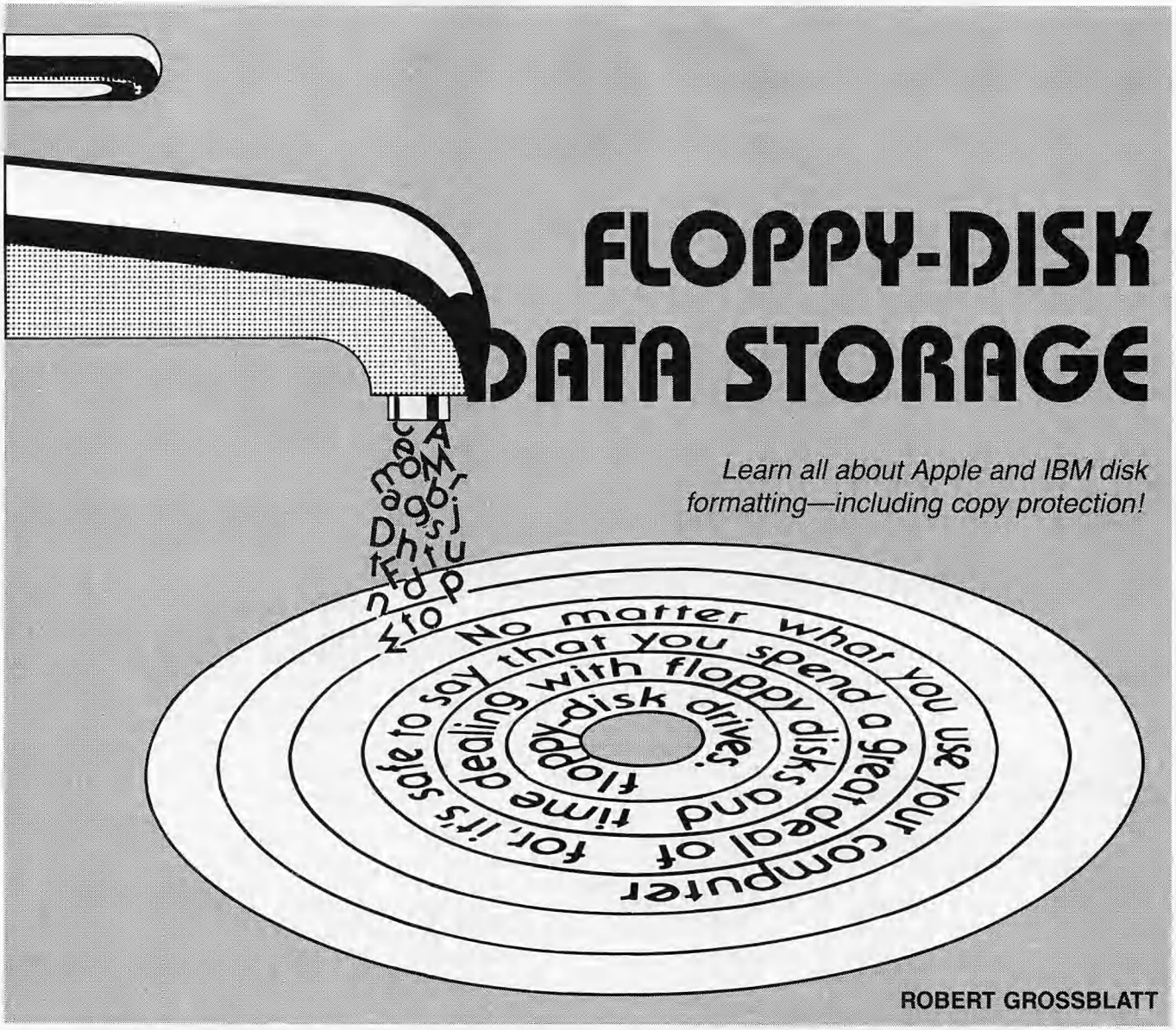
Osborne/McGraw Hill books have for many years provided useful text and reference books for those involved in computers and electronics. Now the book-publishing company has teamed up with Borland International, publisher of many high-quality applications programs and programming languages, including Turbo Pascal, which singlehandedly changed the programming language market when it was introduced several years ago. The two companies have launched a new line of books intended to support many Borland products, including the programming languages Turbo Pascal, Turbo C, and Turbo Prolog, and the applications programs Sprint and Reflex, a word processor and a database manager, respectively.

Advanced Turbo Pascal, by Herbert Schildt, for example, contains twelve chapters on advanced programming techniques in that language, including sorting and searching; queues, stacks, linked lists, and trees; dynamic memory allocation; assembly-language and DOS interfacing; etc. An introductory chapter reviews Pascal basics, and an appendix discusses converting programs from other languages. Schildt's style is quite readable, and his coverage appears thorough. The book contains many example programs, which Schildt will provide on disk for the healthy sum of $24.95.

Products Reviewed

- Microsoft Bookshelf, Microsoft Corporation, 16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717, (206) 882-8080.
  CIRCLE 000 ON FREE INFORMATION CARD

- Osborne/McGraw-Hill, 2600 Tenth Street, Berkeley, CA 94710, (800) 227-0900, (800) 772-2531 (CA).
  CIRCLE 000 ON FREE INFORMATION CARD


No matter what you use your computer for, it's safe to say that you spend a great deal of time dealing with floppy disks and floppy disk drives.

FLOPPY-DISK DATA STORAGE

Learn all about Apple and IBM disk formatting—including copy protection!

ROBERT GROSSBLATT

Part 2 This month we'll turn our attention more fully to copy protection, highlighting the schemes used to make Apple and IBM software secure.

Oddball formatting and nibble counting

The first two copy-protection schemes form the basis for most others. The basic idea behind oddball formatting is to format most of a disk in the usual manner, but to format one or more sectors in a unique manner. That way, when DOS first starts to read the disk, it assumes one format. When it runs into the odd sector, it thinks a data error has occurred. However, the copy-protected program itself knows where to expect that sector and how to deal with it.

Nibble counting is based on the fact that no two disk drives are exactly the same. In order for disks to be readable on various machines, DOS has a built-in "tolerance" factor that programmers can use to implement copy protection.

The major difference among drives is the speed at which the disk rotates. The standard is 300 rpm (200 milliseconds per revolution), but a drive only has to be within about 1% of that speed to be usable.

As we saw earlier, each sector on a disk has the room to store the requisite data. The remainder of the space is filled with gap bytes that separate one sector's data area from the next sector's ID area. The number of gap bytes written on a track is a direct function of the drive's speed and the rate at which the disk hardware spits out bits. But bit timing is in microseconds and drive timing is in milliseconds, so it's safe to say that the number of gap bytes depends exclusively on drive speed.

The gap bytes in a particular track, called the *signature track,* are counted, and the sum is stored elsewhere on the disk. Then, when you run the program (or boot the disk), the head is moved to the signature track, the bytes are counted and compared to the original number. If they're not the same, the software knows it's not running on the original disk, and that's what makes nibble counting tough to crack.

Track erase, directory move

The earliest forms of Apple copy protection were feeble by today's standards. For example, erasing one disk track would bring the normal copy program to a screeching halt. And if the erased track was near the

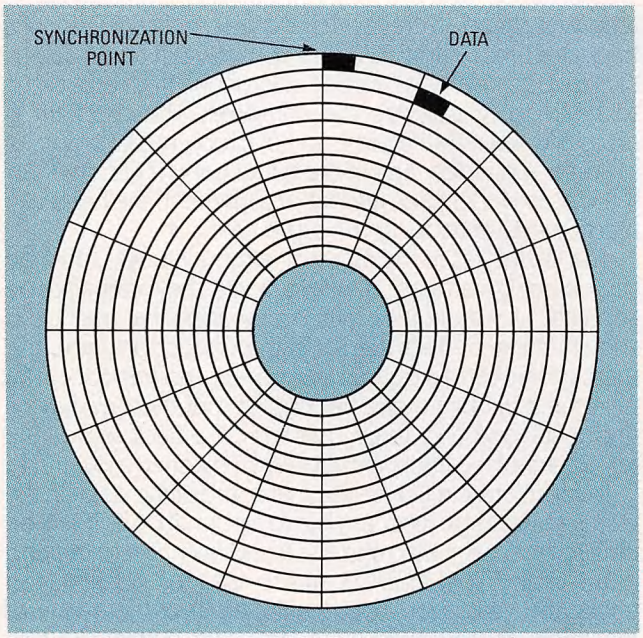


FIG. 5—TRACK SYNCHRONIZATION involves writing data at a particular location, stepping the read/write head, and writing additional data.

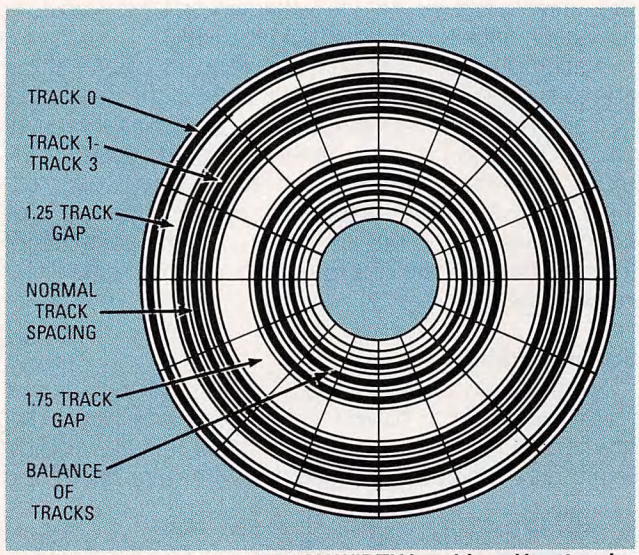


FIG. 6—NON-STANDARD TRACK WIDTH is achieved by stepping the read/write head in ¼-track increments.

outside of the disk (track 3, for example), the rest of the disk would be protected.

Another early method was to move the catalog from track 11, where it was normally found. The files were also made unlistable by changing the standard load locations so you couldn't even look at the files. A reset would wipe out memory and reboot.

Those tricks, and a few others, were attempts to protect programs by altering DOS. They were effective for awhile because no one had taken DOS apart yet, so DOS parameter locations, sector formats, and file structure weren't common knowledge. After those things became known, however, the simple copy-protection schemes were dropped in favor of more-sophisticated ones.

The 36th track

When the copy-protection industry was in its infancy, someone discovered that, although DOS was designed around 35-track hardware, most Apple drives could actually read 36 tracks. That was the first time the disk hardware itself was used as the basis of a copy-protection scheme. Publishers put part of their code on track 36 and checked to make sure it was there whenever the program was run. None of the standard copy programs knew anything about the extra track so the disk was uncopyable—until the method was found out and publicized. Even without the discovery, use of the 36th track was ineffective because not all drives could read that track. So, as soon as consumers started returning software, the method was dropped.

Track syncing

The next type of copy protection can be understood by looking at Fig. 5. When data is either read from or written to a disk, DOS is told to go to a specific track and then look at a particular sector; no special relationship is assumed between adjacent tracks. However, software publishers discovered that it was possible to keep very strict timing relationships when reading the disk. This meant that if you knew where you were on one track at the instant you told the head to step, you knew where you would be when you arrived at the next track.

Building a protection scheme around that fact involves writing a program that reads a track, steps the head when a particular data pattern is found, and then immediately writes some data to the new track. You now have a disk with known data patterns written in a particular order on adjacent tracks. If you change the write to a read and don't find the data pattern you originally wrote there, you know that it's not the original disk.

Playing With Apple DOS

The ways in which you can "customize" DOS are limited only by your imagination. After all, DOS is just another program. Here are a few suggestions on how you can change the internal workings of DOS to provide a measure of copy protection to your own disks.

Normally DOS reads your keyboard commands and tries to execute them. However, by patching in your own routine, you can cause DOS to do just about anything. Location $9FED is a good patch point. In an unmodified DOS, you'll see the following code:

```
9FED    59 A4 A8    EOR    $A884,Y
```

That is the beginning of the code that parses the input line before going into the command table. Here are several ways to patch that code. First, by causing a jump to $C600, any input line will cause the machine to reboot:

```
9FED    4C00 C6    JMP    $C600
```

This line will cause any input line to jump to BASIC:

```
9FED    4C 03 EO    JMP    $E003
```

This line will cause any input line to beep and go into the monitor:

```
9FED    4C 65 FF    JMP    $FF65
```

This line will cause any input line to beep and print "ERR":

```
9FED    4C 2D FF    JMP    $FF2D
```

DOS has both warm and cold boot routines, and it can tell which one is required by looking at the byte stored at location $03F4. If you change the value in DOS, you'll force a cold boot whenever the reset switch is pressed. Normally DOS has the following code:

```
9E36    49 A5    EOR    #$A5
```

By changing the A5 to a 00, you'll be able to initialize a disk with a DOS that will reboot whenever reset is pressed.

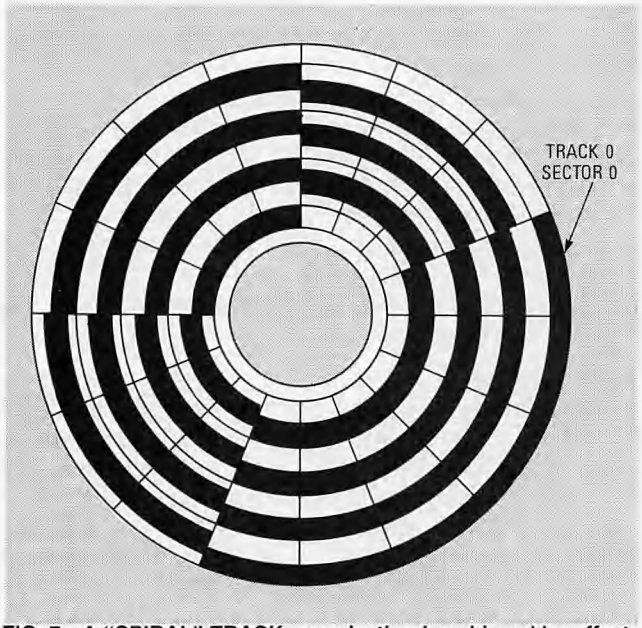


FIG. 7—A "SPIRAL" TRACK organization is achieved by offsetting alternate tracks or groups of tracks.

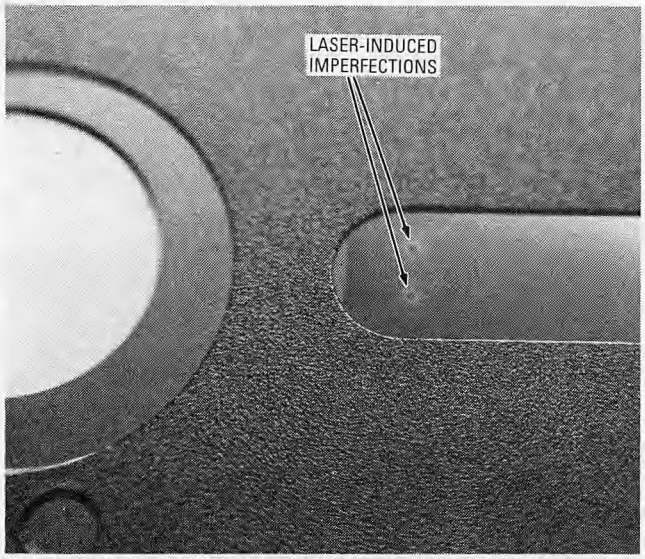


FIG. 8—A LASER-BURNED DISK can be backed up, reformatted, and the copies restored—and the copy-protection is still intact.

Using synchronized tracks became extremely popular. Publishers liked it because it was easy to implement and, at the time, none of the existing copy programs could get around it. Of course, that didn't last long. Most current copy programs can write synchronized copies, so that tracks on the copied disk are arranged in the same order as those on the original disk.

Quarter tracking

All of the copy-protection methods we've discussed so far are techniques that can be implemented from a regular DOS. Apple made their disk system very software intensive, so programmers have much control over the disk hardware. As more and more became known about DOS, programmers found new and sometimes bizarre ways to make their disks unreadable by normal methods.

Some unknown hero in the copy-protection business discovered that Apple DOS didn't actually move the read/write head an entire track at a time, only a quarter of a track at a time. Being able to step between tracks seems useless because data must be at least one track apart for it to be read reliably by the computer. The problem is crosstalk—exactly the same sort of problem that crops up in audio and video tape. If the guard band is too small, or if the recorder's heads are out of alignment, the head can read from two tracks at the same time.

The secret lies in the fact that, although tracks must be separated by at least a whole track, they can also be separated by more than that. A disk using that method is shown in Fig. 6. Track zero is in the correct position, but there's a track and a quarter between it and track one. The next three tracks are one track apart, but then we have a gap that's one and three quarters of a track wide. The remainder of the tracks are separated by one whole track.

It doesn't take much to see that some of the data on the disk is going to be inaccessible to a normal DOS. It will be able to read track zero and the tracks from five to the end of the disk. But when DOS tries to read the odd-spaced tracks, the head won't be positioned over the center of the track, so the signal will be weak. Signal-to-noise problems guarantee that, although some of the data might be read correctly, a good part of it won't. The result is a disk unreadable by any DOS that doesn't know exactly how each track is positioned on the disk.

Writing quarter tracks on a disk requires careful attention to timing details. The stepping rate of the head must be carefully controlled, as does choosing the moment at which data can be read. Many disk drives, particularly older ones with slower stepping rates, have trouble reading a disk with quarter tracks. The problem is more pronounced with tracks that are written near the perimeter of the disk, because the disk turns with a slightly faster linear velocity there.

Spiral tracking

Next, the software industry developed the idea of spiral tracks, which solved the problem of wasted disk space, and made it even more difficult to make copies. Figure 7 is a representation of a disk with spiral tracks. You

LISTING 1

```
10 REM ************************
15 REM * THIS PROGRAM WILL LET *
20 REM * YOU MOVE THE LOCATION *
25 REM * THAT DOS USES FOR THE *
30 REM *    CATALOG TRACK.     *
35 REM ************************
40 :
50 OLDTRACK = PEEK (44033)
90 TEXT : HOME
100 PRINT "THE CATALOG IS NOW ON TRACK ";OLDTRACK
110 PRINT
120 INPUT "WHAT IS THE NEW CATALOG TRACK? ";YOURTRACK
125 VTAB 6: HTAB 1
130 IF YOURTRACK<3 OR YOURTRACK>35 THEN PRINT CHR$(7): GOTO 500
140 :
145 REM PATCH THE FILE MANAGER
150 POKE 44033,YOURTRACK
160 :
165 REM PATCH THE INIT ROUTINE
170 POKE 44703,YOURTRACK: POKE 44764,YOURTRACK
180 :
185 REM PATCH THE CATALOG ROUTINE
190 POKE 46012,YOURTRACK
200 :
210 PRINT "YOU CAN NOW INITIALIZE A DISK THAT HAS"
215 PRINT "FILES UNREADABLE BY A NORMAL DOS."
220 END
500 INVERSE : VTAB 24: HTAB 1
510 PRINT "THE NEW TRACK MUST BE BETWEEN 3 AND 35";: GOTO 125
```

LISTING 2

```
59E4     CE E7 59     DEC $59E7
59E7     CF           ???
59E8     EA           NOP
59E9     59 EF EA     EOR $EAEF,Y
59EC     59 AD 51     EOR $51AD,Y
59EF     C0 AD        CPY #$AD
59F1     54           ???
59F2     C0 AD        CPY #$AD
59F4     57           ???
59F5     C0 AD        CPY #$AD
59F7     52           ???
59F8     C0 20        CPY #$20
59FA     60           RTS
59FB     5B           ???
59FC     20 C5 5B     JSR $5BC5
59FF     20 4E 5B     JSR $5B4E
5A02     A9 04        LDA #$04
5A04     8D EC B7     STA $B7EC
5A07     A9 00        LDA #$00
5A09     8D EB B7     STA $B7EB
5A0C     A9 00        LDA #$00
5A0E     8D F0 B7     STA $B7F0
5A11     A9 60        LDA #$60
5A13     8D F1 B7     STA $B7F1
5A16     A9 40        LDA #$40
5A18     20 45 5A     JSR $5A45
5A1B     10 01        BPL $5A1E
5A1D     A9 20        LDA #$20
5A1F     91 5A        STA ($5A),Y
5A21     AD 50 C0     LDA $C050
5A24     A9 09        LDA #$09
. . .
```

can see that it meets the track-spacing requirement and that it makes maximum use of disk space. Trying to copy a disk like that can result in major brain damage. Not only do you have to know the track spacing, but you also must have the correct pattern.

There are two reasons why spiral tracks are a real problem for a copy program. The first is simply that it's hard to tell how many sectors have been placed on a particular track. The second more serious problem has to do with the nature of copy programs. We've already seen that there are so many ways to protect a disk that a good copy program can't make any assumptions about what it's going to find when it reads the disk. The more it expects to find, the less it will be able to deal with what's really there.

Let's suppose that you've just bought Acme Copy, the roughest, toughest, smartest, copy program in the world—it's so good it can even copy an unformatted disk—and you use it to make a copy of a spiral-tracked disk. You load the program and turn it loose. Even though track zero is written upside down and backwards, Acme Copy copies it without a hitch. Let's also suppose that the rest of the disk is spiral tracked; only half of the sectors on each track contain real data. Acme Copy doesn't know anything about that—no assumptions, remember?

Acme Copy reads in a track, half of which is data and half of which is garbage—and that's where the problem comes in. There is no way for the program to distinguish garbage from copy-protected data. It reads the track's data, does some sort of analysis, writes out the copy, and steps to the next track. Of course the copy will be worthless. Even if Acme Copy goes through the disk quarter track by quarter track, the act of writing a quarter track will undoubtedly corrupt the previous quarter track. It's sad but true that the only way you can get it to work properly is to tell it what spiral pattern to follow as it goes through the disk.

IBM copy-protection

Most of the Apple protection methods we've looked at have their counterpart in the IBM world. Modifying DOS and messing around with sector information were done early on in PC history—and neither method lasted any longer there than in the Apple world. Most IBM copy programs can deal with those methods without even working up a sweat.

However, some of the more imaginative copy-protection methods found on the Apple simply couldn't be ported over to the IBM because of the basic difference in their disk systems. The PC's designers decided to let most of the disk system be handled by an LSI controller IC. That made it simpler to develop DOS, because the controller has built-in routines to handle disk primitives like moving the head, reading and writing data, formatting, and so on. The ability to do quarter tracking, for example, is impossible because the controller hardware can only step the read/write head in full-track increments. In fact, because of the limited repertoire of commands built into the floppy-disk controller IC, just about the only trick that appeared had to do with the index hole.

Contrary to popular belief, the IBM only uses the index hole when formatting a disk. Every time the head is stepped out to a new track, the PC waits for the index hole to appear and uses that point as the starting point for formatting the track. After a disk has been formatted, particular tracks and sectors are located using the same method as the Apple. The floppy controller reads in the sector address of its current position and then steps in or out to the track DOS wants it to read.

Some protection methods want the track-splice point (the place where start and end points meet) to be exactly at the index mark. As with any protection method, however, this one only baffled copy programs for awhile. After the method was uncovered, it wasn't long before most copy programs could handle it.

Undocumented op-codes

IBM's floppy controller is an NEC PD765, which is really a microprocessor that has been optimized to handle disk drives. Some programmers disassembled the microcode in the IC looking for features and abilities that didn't appear in the documentation.

Several undocumented features were found, the most popular of which was to mix FM (frequency modulation)

and MFM (modified frequency modulation) formatting on one track. None of the copy programs were able to handle that mixed formatting, because they didn't know how to make the PD765 do the trick. The result was a nearly unbeatable protection scheme.

However, there are two big problems with undocumented op-codes. The first is that the manufacturers who second-source the IC don't know about them. The second is that, because they aren't part of the 765's published vocabulary, there's no way to guarantee they'll still be there when new versions of the IC are released. And that's what spelled the death of mixed formatting on the IBM. The software ran well on computers that used the same run of 765's, but died on other machines. Needless to say, the scheme was dropped.

Other methods

The search is always on to discover new and wonderful ways to lock up disks. The older, software-only methods such as altering DOS or playing around with sector formatting and address bytes, are still used because they're inexpensive and easy to do. They're usually found on games and other low-priced software. Even though most copy programs know how to deal with them, some are still hard to beat, particularly nibble counting.

The publishers of more expensive software, however, have deep enough pockets to be able to afford more expensive protection schemes. There are two major high-end methods for protecting disks. Both are expensive because it takes more than just changing a few bytes to get them on a disk. The first involves what are called "weak bits," sectors written in such a way that they don't read the same way twice. All that's needed to activate weak-bit protection is to do two successive reads. If they're the same, the software knows it's running on a copy and can take appropriate action. Getting a sector like that on a disk involves the use of special duplicating equipment; it can't be done with a stock PD765.

Just as weak bits make the disk unique from a formatting point of view, laser holes make it unique from a physical point of view. The word *hole* is misleading because the disk isn't actually punctured, but burned, usually in two different places. The photograph in Fig. 8 shows what to look for if you suspect that the disk you're trying to copy has that kind of protection. The marks are usually located on the back side of the disk near the hub. It's easy for the software to check for the laser holes.

Laser-treated disks are probably the most expensive form of copy protection, but they have one big advantage for a publisher. Because the disk is physically unique, the publisher doesn't have to protect the files and can let you back them up on a regular disk. If you develop an error on the original disk, you can reformat it and copy the files from your backup. Remember that reformatting the disk has no effect on the laser holes. However, most copy programs can even get around laser-treated disks.

Because of the ease of overcoming most disk-based schemes, state-of-the-art copy protection these days is the hardware lock. It's a device that plugs into your serial or parallel port and remains totally transparent (in theory, at least) to the normal operation of the port. Software can check whether the device is present at various times during execution, and come to a screeching halt if it doesn't get the proper response.

References

Books (Apple): *Beneath Apple DOS,* Don Worth & Pieter Lechner, Quality Software, Computer Book Division, 21601 Marilla Street, Chatsworth, CA 91311, (818) 709-1721. *Beneath Apple ProDOS,* Don Worth & Pieter Lechner, Quality Software (address above).

Books (IBM): *The Programmer's Guide to the IBM PC,* Peter Norton, Microsoft Press, Dept. RC06, Box 97200, 10700 Northup Way, Bellevue, WA 98009. *Advanced MS-DOS,* Ray Duncan, Microsoft Press (address above).

Hardware (IBM only): The Copy II PC Option Board, Central Point Software, 9700 SW Capitol Highway, Suite 100, Portland, OR 97219, (503) 244-5782.

Software (Apple): *Locksmith 6.0,* Alpha Logic Business Systems, 4119 North Union Road, Woodstock IL, 60098, (815) 568-5166. *Copy II Plus,* Central Point Software, (address above). *Bag of Tricks 2,* Quality Software (address above). *Disk Repair Kit,* Penguin Software, 830 4th Avenue, P.O. Box 311, Geneva, IL 60134, (312) 232-1984.

Software (IBM): *Copy II PC,* Central Point Software (address above). *Master Key,* Sharpe Systems, Corp., 2320 E. Street, La Verne, CA 91750, (714) 596-0070. *CopyWrite,* Quaid Software Limited, 45 Charles Street East, Department 740, Third Floor, Toronto, Ontario M4Y 1S2, (416) 961-8243.

However, hardware locks are very expensive, so it's unlikely you'll see one protecting inexpensive software. Can a hardware lock be beaten? The answer is yes—sort of. The qualifier is there because, no matter what kind of protection scheme is employed, it only protects the disk, not the data. All the fancy tricks that have been used to lock up the software fall away when the program is loaded into the computer.

Your own copy protection

Several of the books mentioned in the References sidebar contain complete discussions and disassemblies of various versions of Apple DOS. By studying that information, you'll see that there are several ways to alter DOS and add simple copy protection to your own disks. DOS commands can be changed (or eliminated altogether), or your own code can be inserted in one of the unused areas of DOS. Then, when your program goes looking for it, if the code is there, the program will run normally. If it's not there—well, the choice is yours. You could be kind and just reboot the system, or you could be nasty and trash a couple of tracks.

If you're interested in playing with a modified disk organization, initialize a spare disk and run the program shown in Listing 1. It will create a diskette with the catalog located in a non-standard location.

To create a custom version of DOS, you can play with the ideas shown in the sidebar entitled Playing With DOS. However, just remember that those are fairly simple schemes, and they'll be no real obstacle to someone whose primary mission in life is to get a look at your code. The kinds of things you can do by messing around with DOS are the things a real "crack-ist" eats for lunch.

continued on page 102

BUILD THE PT-68K

Address decoders, RAM, and ROM.

© 1987, Peter A. Stark

PETER A. STARK, STARK SOFTWARE SYSTEMS CORPORATION

PART 4 In last month's installment, we described the 68000 microprocessor and got it running. This time we continue with the address decoders, RAM, and ROM. We'll assume that the jumpers we installed last time are still installed; don't remove those jumpers until you are told to do so. (One jumper connects pins 14 and 22 of IC47, to negate the 68000's $\overline{\text{BERR}}$ line, and one connects pins 1 and 14 of IC66, to assert the 68000's $\overline{\text{DTACK}}$ line.) In addition, the data bus should still be grounded via the two sets of Molex pins installed in the sockets for IC21 and IC27.

Step 8: Address decoding

Before building the address decoder, we need to develop the $\overline{\text{MAP}}$ signal, which provides one input to the decoder. When the 68000 starts operating after power is applied or after a reset, it must know (a) where to place its stack, and (b) where to start executing instruction. It looks for those in the first eight bytes of memory, starting at address $000000.

The problem is that in most 68000 computers address $000000 is in RAM, and that RAM will contain random values after the computer is first turned on. So how and where does the 68000 find the appropriate addresses? The usual solution is to set up the address decoder so that ROM, not RAM, occupies address $000000 and beyond. The trick is that the ROM remains there just long enough for the 68000 to get its two addresses, after which the ROM is switched out of and the RAM is switched into the low-order addresses. A ROM of that type is often called a *phantom ROM* or a *shadow ROM* because it's seemingly there and then gone. The circuit in Fig. 1 implements the ROM-switching function.

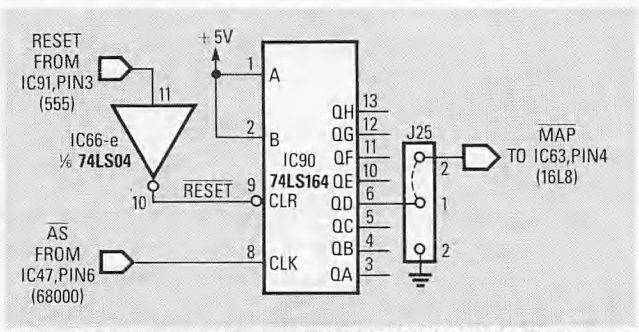


FIG. 1—THIS CIRCUIT GENERATES THE MAP SIGNAL that forces the microprocessor to address ROM rather than RAM at power up and on subsequent resets.

IC90 is an eight-stage shift register. When power is first applied to the computer, and each time it is reset, all the flip-flops in IC90 are cleared by the $\overline{\text{RESET}}$ signal. The output of the forth stage (Q_D), which is jumpered through J25, then provides a low $\overline{\text{MAP}}$ signal that tells the address decoder to put the ROM at address $000000.

When the 68000 starts, it fetches the stack address and the program start address from ROM in four 16-bit reads; each read is accompanied by an address strobe ($\overline{\text{AS}}$). The positive (trailing) edge of each $\overline{\text{AS}}$ then clocks IC90. Since both shift register inputs are tied high, each clock pulse shifts a high further into the register. After the first clock pulse, that high gets to Q_A; after the second pulse it gets to Q_B; and so on. After the fourth pulse it gets to Q_D and negates $\overline{\text{MAP}}$, which tells the address decoder to disconnect the ROM from address $000000 and substitute RAM.

Now that we understand how the circuit actually works, let's build it and verify that it works. First install a socket for IC90 and a three-pin header block at J25. Insert the IC and place a shorting plug over position 1 of J25. (IC66 was installed in a previous step.) Turn on the power and verify that the $\overline{\text{MAP}}$ signal on the center pin of J25 goes low while you short the reset pins (J23), and that it goes high about a second later, at the same time as the HALT LED goes off. (Unless you have a high-quality oscilloscope, it may be difficult to verify that exactly four $\overline{\text{AS}}$ pulses go by before $\overline{\text{MAP}}$ goes high, so you'll have to take it on faith that the circuit is functional.)

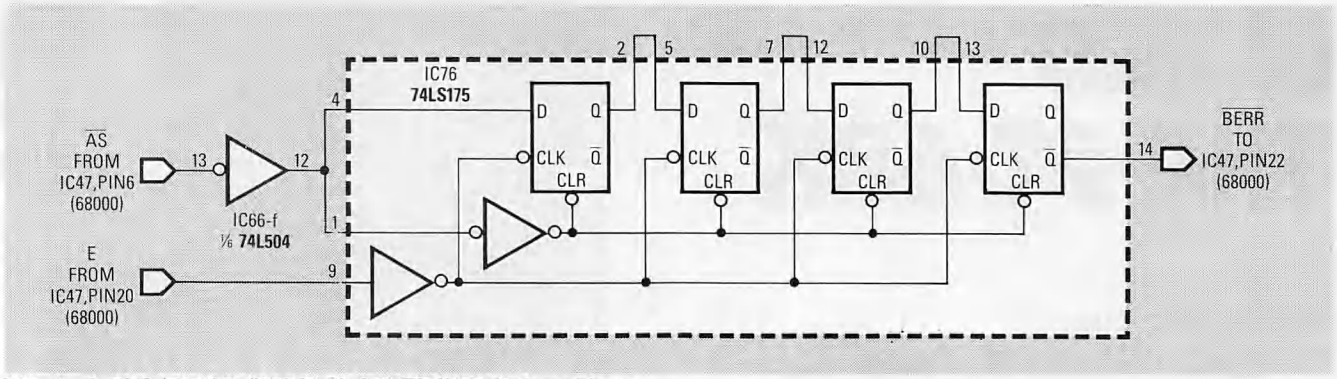


FIG. 2—A BUS ERROR (BERR) IS GENERATED if the address strobe (AS) continues through four E cycles without a data acknowledge (DTACK).

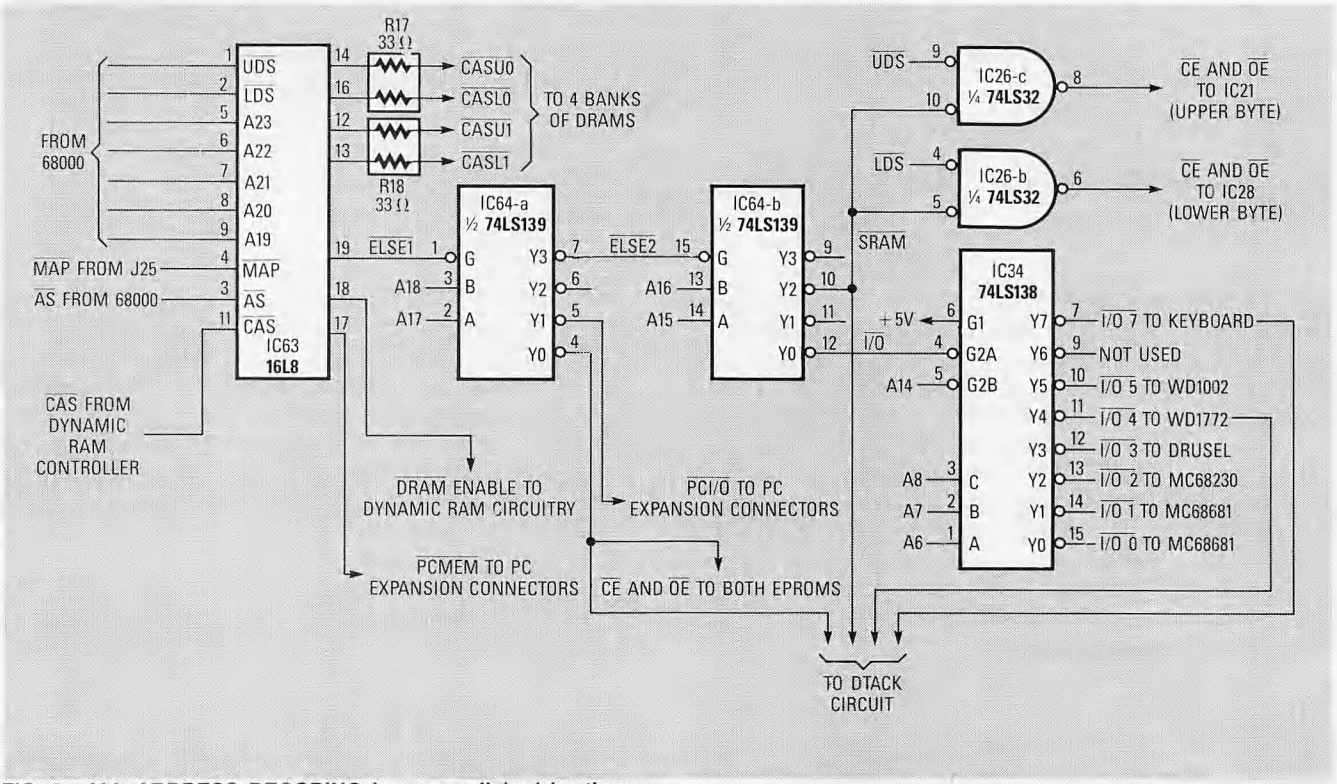


FIG. 3—ALL ADDRESS DECODING is accomplished by three IC's: each IC (or section thereof) further narrows the range of addresses decoded by its predecessor.

Step 9: The bus error circuit

Each time the 68000 wants to access memory or I/O, it sends out an address; LDS, UDS, or both; R/W; and AS. Then it sits back and waits for the memory or I/O to respond. If all goes well, the external circuit should return a low on DTACK (data transfer acknowledge); if something goes wrong, it should return a low on BERR (bus error).

When using slow memory or I/O, DTACK can be used to slow down the 68000; the period of waiting is called a *wait state*. But suppose the 68000 tries—accidentally or otherwise—to access an address at which there is no memory or I/O. What happens then? Since there is nothing to generate DTACK, the 68000 might go into a permanent wait state. That's where the bus-error circuit comes in. Its job is to detect the lack of DTACK after some period of time, and then generate BERR, which forces the 68000 into an error-recovery procedure.

When we first powered the 68000 up last time, we tied DTACK low and BERR high, which allowed the processor to run at maximum speed, and made it think that a bus error never occurred. Since we are still providing a fake DTACK, the 68000 will go full-speed ahead, even though there is no memory in the system yet. However, we may as well install the BERR circuit at this time.

The circuit shown in Fig. 2 looks complex, but actually it consists of just one new part: IC76, a 74LS175 quad D flip-flop. (IC66 was installed in a previous step).

The four flip-flops are configured as a four-stage shift register that is driven by the 68000's E clock. That signal operates at one tenth the speed of MPUCLK, or 800 kHz if the main clock is 8 MHz.

The input data comes from AS, which is also connected to IC76's CLEAR pin. When the 68000 starts a memory or I/O access, it asserts AS, which simultaneously negates IC76's

BUBBLES AND GATES

A given signal in a digital system can be either active high or active low; the signal's name often indicates which. For example, $\overline{\text{DTACK}}$ is active low, and we know it as such because of the bar over the name. FCO, on the other hand, is active high because it has no bar above it.

One way to mark a signal in a schematic diagram is with a *bubble*, which is simply a small circle at the end of a connecting line. Bubbles are a notational convention that, when used properly, can help prevent design errors. The basic idea is that active-high outputs can only go to active-high inputs; likewise with active-low signals. Many engineers and technicians tend to get careless with bubbles, so diagrams can't always be trusted. However, when properly used, bubbles can be very helpful. Look at Fig. 1 for a simple example; the circuit shown there is the Halt indicator discussed in an earlier installment.

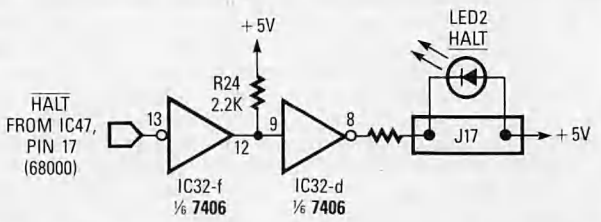


FIG. 1

The $\overline{\text{HALT}}$ signal comes from pin 17 of the 68000 microprocessor; that signal goes low when the 68000 is halted. As stated above, an active-low output must go to an active-low input, so you'll notice a bubble on the input (pin 13) of IC32-f. The output of that inverter is active high, so it goes to an active-high input, IC32-d. The output of that inverter is active low, so we would expect that the LED would light when that output was low, and that is indeed the case.

It's important to understand that IC32d and IC32f are functionally identical—they're both open-collector inverters—even though their symbols are different.

The same principles can be applied to all the basic logic elements, as shown in Fig. 2. Each pair in that diagram is equivalent. For example, the gate on the left in Fig. 2-*a* is normally called an AND gate; described in words, its job is to "make the output high if input A is high *and* input B is high."

There is another way of describing its operation: If either input is low, then the output is low. So we could say "make the output low if A is low *or* B is low." That sounds like an OR gate that works with low-going signals—and that's what the gate on the right in Fig. 2-*a* shows.

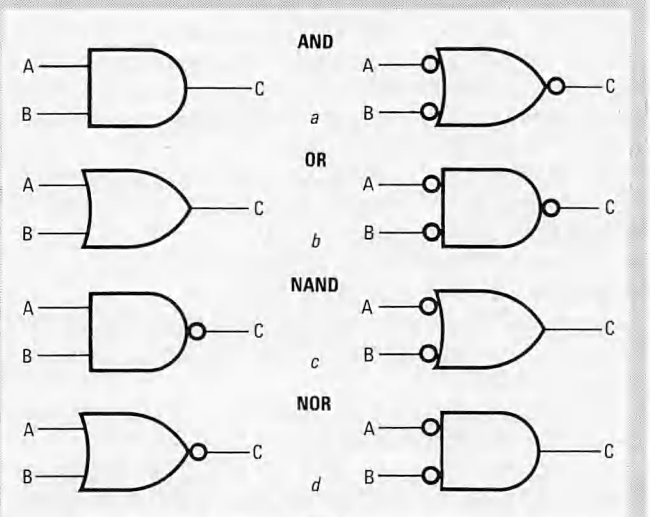


FIG. 2

You can convert from one form to the other by negating all terms on both sides of a logic equation. Highs become lows and vice versa, AND's become OR's and vice versa, and NAND's become NOR's and vice versa. So

$$A \text{ AND } B = C$$

becomes

$$\overline{A} \text{ or } \overline{B} = \overline{C}$$

A designer might use either symbol for the same device, depending on what a particular circuit is supposed to do. Parts manuals, for example, usually describe gates in terms of the "positive logic" function—the symbols on the left in Fig. 2. The 7408, for example, may be described as a "quadruple two-input positive-AND gate." That means the 7408 has four two-input gates, each of which functions as a normal AND gate. However, the 7408 could just as well be called a "quadruple two-input negative-OR gate.

Similarly, Fig. 2-*b* shows a positive-logic OR gate on the left, and a negative-logic AND gate on the right. In reality, both functions could be provided by the same IC (a 7432).

A real-life example is the $\overline{\text{DTACK}}$ generator shown in Fig. 5 of this installment. Note IC36—it's normally described as an eight-input NAND gate, but here it functions as an eight-input NOR gate.

All of that may sound strange, but as we go on, you'll see that the new symbols we've outlined here will help explain how the PT-68K computer works.

CLR input and asserts its D input. Then, each time an E clock signal arrives, that high is shifted one stage to the right. If the $\overline{\text{AS}}$ signal continues through four E cycles (about 4–5 microseconds), the last flip-flop will set, and $\overline{\text{BERR}}$ will go low, informing the 68000 that too much time has passed since the memory or I/O access started. At that point a special interrupt occurs, and a software routine must decide what to do next.

Building the circuit is easy. First remove the jumper that was installed between pins 14 and 22 of IC47. (That jumper forced $\overline{\text{BERR}}$ high.) Next install a socket for IC76, and then the IC (a 74LS175).

Testing is almost as easy. Using the LED probe, look at pins 4 and 9 of IC76. Because $\overline{\text{AS}}$ and E are both square waves, and are on about 50% of the time, the LED should glow at about half brightness. Then look at pin 2, the Q output of the first flip-flop. $\overline{\text{AS}}$ and E are not in any particular phase relationship, so the first flip-flop will trigger once every few $\overline{\text{AS}}$ cycles, but it will never stay on for very long. Therefore, the flip-flop is mostly off, so the LED should be quite dim. The other flip-flops never get a chance to set, so the LED should remain dark when testing any of the other Q outputs. $\overline{\text{BERR}}$, of course, is high all the time, so the LED should be bright when testing pin 14.

Step 10: The address decoder

As described way back in the first part of this series, the address decoder continuously monitors the high order bits of the address bus and signals the RAM, the ROM, and the I/O interfaces whenever an address comes along that is intended for them. The PT-68K's address decoder consists of three IC's, as shown in Fig. 3.

The heart of the decoder is IC63, a 16L8 PLD (programmable logic device). A PLD is like a fast ROM in that it has a number of input and output lines. Each time a combination of ones and zeroes is presented to its input lines, the

PLD outputs a number that is stored in the corresponding location in its internal memory.

By the way, there are significant differences between a ROM and a PLD. The ROM is more complex than the PLD because it can store more information; but, being simpler, the PLD can run faster. In general, the ROM is meant to store numbers, and the PLD is meant to replace logic IC's. In our case, the PLD replaces almost a dozen gates and inverters, thereby saving both space and money.

The PLD functions as follows. It splits the sixteen-megabyte address space of the computer into three areas:

- $000000–$0FFFFF, system DRAM
- $C00000–$DFFFFF, PC-compatible slots
- $F80000–$FFFFFF, everything else

Any other address simply isn't recognized by the address decoder. Now let's examine the details of how the circuit actually works.

1. If $\overline{AS}$ is negated (high), the PLD does nothing; $\overline{AS}$ must be low to ensure that a valid address exists.

2. If address lines A20–A23 are low, representing addresses $000000 through $0FFFFF (i. e., the first megabyte of memory, where the system DRAM resides), pin 18 ($\overline{DRAM}$) is asserted. In addition, if $\overline{CAS}$ is low, indicating that the DRAM refresh circuitry wants to access a column of DRAM, IC63 asserts one or two of the $\overline{CAS}$ outputs (pins 12, 13, 14, and 16), which in turn activate a group of dynamic RAM IC's. $\overline{LDS}$ and $\overline{UDS}$ determine whether to assert one of the lower ($\overline{CASL}$) outputs, one of the upper ($\overline{CASU}$) outputs, or both, depending on whether an odd byte, an even byte, or a sixteen-bit word is to be accessed. Meanwhile, the A19 input of IC63 splits the DRAM megabyte into lower and upper 512K groups.

Incidentally, the four resistors, parts of 33-ohm resistor packs R17 and R18, slow the rise and fall times of those signals. Each $\overline{CASL}$ and $\overline{CASU}$ signal goes to eight DRAM IC's, and the fast rise and fall times cause sharp signal edges, which contribute to noise in the memory. The resistors reduce that noise and thereby improve reliability.

3. If the three high-order bits of the address bus are 110, representing all addresses beginning with a hex C (1100) or hex D (1101), $\overline{PCMEM}$ is asserted. That signal goes to the six PC-compatible expansion connectors for accessing memory on plug-in cards.

4. If the five high-order address lines are high, representing all addresses between $F80000 and $FFFFFF, $\overline{ELSE1}$ is asserted; that signal implies that the chosen address contains something besides system RAM or memory in an expansion slot.

5. Operation is different when $\overline{MAP}$ is low; then the DRAM is disabled and $\overline{ELSE1}$ signal is asserted instead. That maps the ROM instead of the RAM into address $000000.

I/O decoding

When $\overline{ELSE1}$ is low, the address space from $F80000–$FFFFFF is divided into three groups by IC64-a, according to the states of A18 and A17. When the $\overline{G}$ input of IC64 is asserted, one output is asserted according to the binary states of the A and B inputs. For example, the $\overline{Y0}$ output is asserted when both inputs are low.

1. When A18 and A17 are 00, which occurs for addresses $F80000–$F9FFFF, IC64-a asserts the $\overline{CE}$ and $\overline{OE}$ signals for both EPROM's, thereby enabling them.

2. When A18 and A17 are 01, which occurs for address $FA0000–$FBFFFF, IC64-a asserts $\overline{PCI/O}$, which goes to the six PC-compatible expansion connectors for accessing port-mapped I/O on plug-in cards.

3. When A18 and A17 are 10, which occurs for address $FF0000–$FDFFFF, IC64-a asserts its $\overline{Y2}$ output, which is unconnected.

4. When A18 and A17 are 11, which occurs for address $FE0000–$FFFFFF, IC64-a asserts $\overline{ELSE2}$, which enables another decoder, IC64-b. So, as you can see, each IC in the address-decoding chain uses the output of its predecessor in the chain to narrow down the range of addresses it recognizes. In this case, IC64-b splits the $FE0000–$FFFFFF range into two smaller groups, depending on address lines A16 and A15.

1. If A16 and A15 are 00, which occurs for addresses $FE0000–$FE7FFF, IC64-b asserts $\overline{I/O}$, which drives IC34.

2. If A16 and A15 are 10, which occurs for addresses $FF0000–$FF7FFF, IC64-b asserts $\overline{SRAM}$, which drives the static RAM and the clock/calendar via IC26-b and IC26-c. (The RAM circuit was shown in Fig. 3 in the December installment.) Note that $\overline{LDS}$ and $\overline{UDS}$ are combined with $\overline{SRAM}$ to enable the upper byte (IC21), the lower byte (IC28), or both. For example, when both $\overline{UDS}$ and $\overline{SRAM}$ are asserted, the $\overline{CE}$ and $\overline{OE}$ pins of IC21 will be asserted, and the IC will be enabled.

WHAT'S IT ALL ABOUT?

The computer described in this series of articles is based on Motorola's 68000 microprocessor, which is also used in a number of personal computers (including the Macintosh, the Atari ST, the Commodore Amiga) engineering workstations, and laser printers. The computer, dubbed the PT-68K, will not run software designed for the IBM PC—at least not yet. But it will accept PC hardware, including keyboard, video display card, monitor, I/O ports, etc.

The PT-68K has a monitor program (HUMBUG) and a version of BASIC, both of which are contained in on-board EPROM. By adding a disk drive and the appropriate control circuitry, the computer will run the SK*DOS operating system, which includes an editor, an assembler, numerous utilities, and a 6809 emulator, which allows you to run much already written 6809 software. In addition, development of a full C compiler and an extended BASIC are nearly complete, and several firms have announced products including a full-screen editor, a disassembler, a communications program, and more. Up-to-date information on software and hardware is available from the SK*DOS BBS (914-241-3307). Technical support is also available there.

The PT-68K motherboard is the same size and shape as an IBM PC XT motherboard, so it may be mounted in an inexpensive clone cabinet. In addition, the PT-68K may be powered by an inexpensive PC clone power supply. Either a serial terminal, or a personal computer running a terminal emulator, or an IBM monochrome adapter card, monitor, and keyboard may be used to communicate with the PT-68K.

Presently we are looking into the possibility of designing or adapting a plug-in card that would contain an Intel microprocessor and would allow you to run PC software on your PT-68K. Right now, it looks very promising, and we'll keep you up to date on our progress.

ERROR CORRECTIONS

- The parts list presented in the October issue listed an incorrect part number for IC28; the correct number is 74S74.
- Several parts were inadvertently omitted from the parts list, including LED1–LED3, SPKR1, and 0.1 μF bypass capacitors C69–C73.
- The parts-placement diagram (Fig. 3 in the November installment) was printed upside down, so references in the text will be rotated 180° from the true position. However, all locations on the board are labeled correctly.

3. The 01 and 11 states of A16 and A15 are not used.

That brings us to IC34, the last IC in the address-decoding chain. IC34's job is to decode addresses of the seven on-board system I/O devices: the two MC68681 DUART (dual UART) serial IC's, the MC68230 parallel port, a drive select latch that controls the floppy-disk drive, the WD1772 floppy-disk controller IC, an optional WD1002 hard-disk controller, and the PC-compatible keyboard. As stated above, IC34 is enabled by the $I/\overline{O}$ signal that is asserted for addresses $FE0000–$FE7FFF. In addition, IC34 requires a low (at its $\overline{G2B}$ input) from A14, so it only responds to the 16K of addresses in the range from $FE0000–$FE3FFF. Then it decodes address lines A6–A8, asserting one of the $\overline{Y0–Y7}$ outputs. For example, when A8, A7, and A6 are 000, IC34 asserts the $\overline{I/O0}$ line, which enables the first DUART.

Now comes a difficult question: What is the range of addresses that the DUART responds to? Following the circuit from left to right, we see that, for $\overline{I/O0}$ to be low:

- A23 through A19 must be 11000
- A18 through A14 must be 00000
- A13 through A9 are unknown
- A8 through A6 must be 000
- A5 through A0 are unknown.

It's evident that the address decoder does not really look at all the bits of the address bus—there are eleven bits that are unaccounted for (or ten bits when we realize that A0 doesn't exist.) Let's ignore A0–A5 for the moment; A9–A13 are the problem.

Grouping those bits left to right by fours, they look like this:

1111 1110 00xx xxx0 00yy yyyy

Each of the five unknown bits corresponding to A9–A13 is labeled with an *x*; each of the six unknown bits corresponding to A0–A5 is labeled with a *y*. Each x and y bit could be either a zero or a one; first let's assume that all the *x* bits are zeroes:

1111 1110 0000 0000 00yy yyyy

Converting to hexadecimal, the first four digits are clearly $F800. If the *y* bits are all zeroes, then the last two digits are $00; if they are all ones, then the last two digits are $3F. That tells us that the DUART is addressed from $F80000–$F8003F, a total of 64 locations.

But the *x* digits need not all be zeroes; they could just as well be 00001, which would make the complete address look like this:

1111 1110 0000 0010 00yy yyyy

The first four digits here are $FE02; again assuming that the *y* digits can be anything from all zeroes to all ones, that gives us an address range of $FE0200 through $FE023F. In a similar manner, we see that the DUART responds to many addresses:

- $FE0000–$FE003F
- $FE0200–$FE023F
- $FE0400–$FE043F
- $FE0600–$FE063F
- $FE0800–$FE083F
- ... through ...
- $FE3E00–$FE3E3F

The latter addresses are decoded when the *x* bits are all ones.

The preceding is an example of *incomplete address decoding;* it arises because the address decoder does not decode all bits. In fact, the bits that can be either value are usually called *don't cares.* Incomplete address decoding results in using up more addresses than are actually needed, but it allows the circuitry to be simpler and less expensive. When programming an incompletely decoded device, we usually work with the base location—the 64 locations from $FE0000–$FE003F in this particular case.

The EPROM's are also incompletely decoded. As stated above, the EPROM is assigned addresses from $F80000 through $F9FFFF, a total of 128K bytes. Typically, however, a pair of 27128's would be used, and they provide a total of only 32K, so 96K of address space is wasted. What actually happens is that the 32K of EPROM appears in the 128K space four times. That is, the EPROM *appears* to occupy the entire 128K, but on closer examination we see that there are four copies of the same data in that space. The loss of that many addresses in an 8-bit computer with a total of 64K of addresses would be unthinkable; in a computer with 16 megabytes of addresses, the loss is insignificant.

The same situation occurs with the I/O. Although each I/O device may only require a few bytes of address space, each takes up 32 64-byte chunks of addresses, for a total of 2048 addresses.

Assemble and test the address decoder

Enough theory; it's time to build the address decoder. Install the following components: IC63 (with socket), R17 and R18 (solder directly to the board), IC64 (with socket), IC34 (with socket), and 0.1 μF capacitors at C13, C67, C68, and C70. Place the J25 jumper in position 1, recheck all connections, and then apply power.

Although a thorough test of the address decoder would require sophisticated test equipment, we can perform a simple test with the aid of Fig. 4, which shows the waveforms output by the decoder circuit.

As you recall from last time, while our test circuit is wired up (i. e., the data bus is shorted to ground) the 68000 executes what it thinks are four million OR instructions, looping through the entire 16-megabyte memory space once every four seconds. Figure 4 shows what happens at several key points. During those four seconds the 68000 counts up through 16 megabytes of memory. The left edge of the figure corresponds to memory location $000000, the right edge (just before the pulses) to location $FFFFFF. The pulses at the extreme right end of the figure indicate the beginning of the next four-second cycle.

First let's look at the $\overline{DRAM}$ signal. Since dynamic RAM occupies the first megabyte of addresses, the $\overline{DRAM}$ signal

Ordering Information

Complete details were given in part one (in the October issue). To summarize: The basic kit (PT1, $200) contains all parts except power supply, case, and video terminal or personal computer to get a small system (ROM monitor, 2K RAM) up and running. The full basic system (PT68K, $460) includes 512K of dynamic RAM, floppy-disk controller, parallel port, battery-backed clock/calendar, and three PC-compatible expansion slots. To order, or for more information, contact Peripheral Technology, 1480 Terrell Mill Road #870, Marietta, GA 30067, (404) 984-0742.

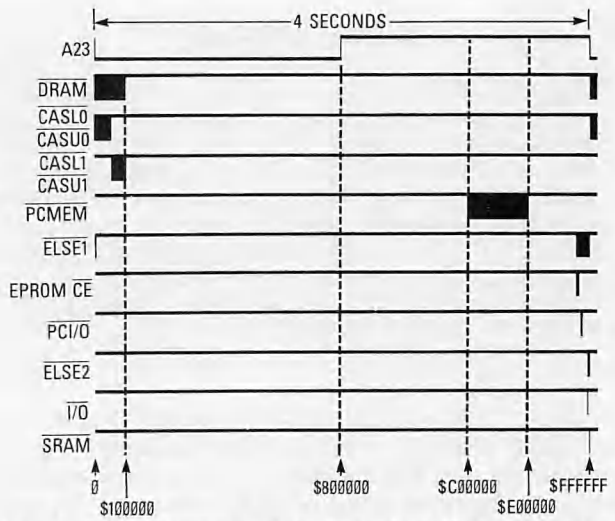


FIG. 4—VARIOUS OUTPUTS OF THE ADDRESS DECODER are shown here; these waveforms will be seen only with the test circuit (shorted data bus, etc.) installed.

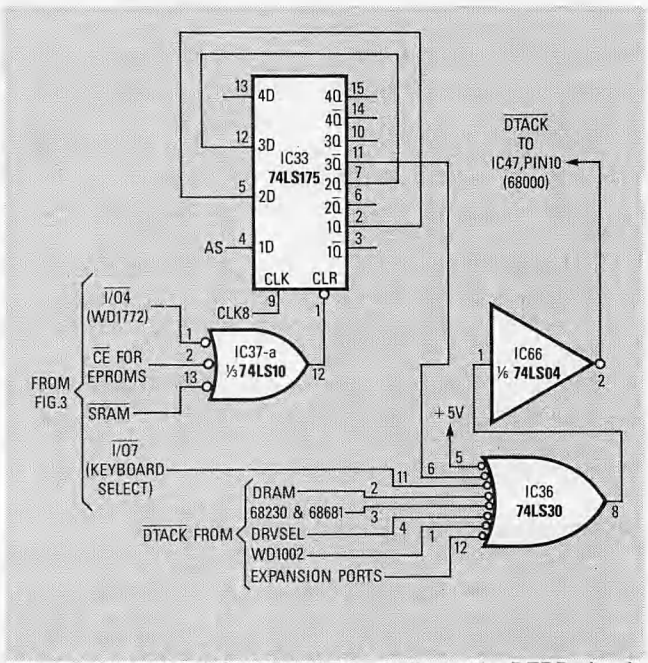


FIG. 5—THE DTACK GENERATOR is similar to the BERR circuit shown in Fig. 2. Here, however, the shift register (IC33) prevents the computer from locking up should a memory or I/O device not supply a data acknowledge.

goes low for the first 1/16th of the four-second cycle, or 1/4 second. The DRAM signal does not remain low for the entire 1/4 second, however, and that's why it's shown as a solid block in Fig. 4. It doesn't stay low because, like all other signals in the address decoder, DRAM is asserted only when the AS is asserted. Because AS rapidly cycles between high and low, so does DRAM.

You can verify that with a good-quality scope; you should see a continuous high, with a short burst of pulses once every four seconds. Without a scope, you can you connect your LED probe to DRAM (at pin 18 of IC63). The LED should remain on continucusly (because the signal is normally high), but once every four seconds, the LED will get slightly dimmer for about a quarter of a second.

Each of the other address-decoder outputs can be checked in the same way, except for the four CAS outputs, which will show a constant high because we have not yet supplied a CAS input to IC63. Some of the other outputs will remain on for a longer period of time—such as PCMEM, which will dim the LED for about 1/2 second—but most will be much shorter. The I/O and SRAM outputs will be very difficult to see since they last only about ten milliseconds, just barely long enough to flicker the LED if you watch carefully. The outputs of IC34 are too short to be able to see on the LED, although a pulse-catching logic probe or oscilloscope will show a slight flicker once every four seconds.

Step 11: The data acknowledge circuit

Previously we generated an artificial DTACK with a jumper; now it's time to install the real DTACK circuit and see how it works.

Ideally, every device, whether memory or I/O, should generate its own DTACK when it finishes an operation. The 68000 would then know that it was time to continue. That is practical in some circuits (DRAM and some I/O devices, for example), but in others a timer is required; the timer just waits for a period time and then generates DTACK, assuming all is well. However, the PT-68K system uses both approaches.

Fig. 5 shows the DTACK generator circuit. It has eleven inputs, of which two (AS and CLK8) are used for timing.

Four other inputs (I/O4, I/O7, the EPROM CE signal, and SRAM) come directly from the address decoder in Fig. 5. I/O4 goes low whenever the address decoder enables the WD1772 floppy-disk controller; CE goes low when either EPROM is selected; and SRAM goes low when the static RAM is selected.

Whenever I/O4, CE, or SRAM goes low, IC37-a provides a high to the CLR input of IC33, a quad D flip-flop, which is configured like IC76 in Fig. 2. IC33 is clocked by CLK8, the 8-MHz clock signal. The output of IC37-a is usually low, so its four flip-flops are usually in the reset state. Therefore, the 3Q output is usually high.

Now suppose that the floppy-disk controller, the EPROM, or the static RAM is decoded. The output of IC37-a will go high, release IC33's CLR signal, and allow it to start shifting. AS is high at that point, and that high starts shifting through the flip-flops. After three CLK8 pulses, 3Q goes low, which sends the output of IC36 high, and the output of IC66-a low, thereby generating DTACK. In other words, IC33 acts as a delay. Note that it is clocked by CLK8, not MPUCLK. Even if you speed up the computer by using a faster MPUCLK, IC33's delay will not change.

Now let's look at IC36. When any of its inputs goes low, its output goes high. One input (pin 5) is not used, so it is tied high. Of the remaining inputs, one comes from IC33, as we saw above; one comes from the keyboard select line (I/O7); and the other five come from other parts of the computer that we'll discuss (and build) in a later installment of this series.

For now, just understand that each of the other sections generates its own data-acknowledge signal, all of which are combined at IC36. In addition, the keyboard circuit operates very fast, so I/O7, its select signal, immediately generates its own data-acknowledge signal.

To build the DTACK circuit, first remove the jumper between pins 1 and 14 of IC66. Then install IC33 and IC36 (with sockets). IC37 and IC66 were installed previously; J25 should still be in position 1 from the previous step;

and the Molex pins should still be inserted in the EPROM sockets (IC21 and IC27). Now turn on the power and let's see what happens.

Nothing! Well, of course not. The problem is that the 68000 is still trying to execute four million OR instructions. It's still getting them from the Molex pins, but it's not getting $\overline{\text{DTACK}}$. Hence the $\overline{\text{BERR}}$ circuit is timing out and halting everything. You probably noticed that the HALT LED goes on and never even flickers.

Now move the jumper at J25 from position 1 to position 2 and try again; you will see a slight flicker on the HALT LED about a second after you turn on the power (or force a reset by shorting J23), but it still goes on.

The Molex pins in IC21 and IC27 put all zeroes on the data bus, so the address that the 68000 picks up after a reset is also all zeroes. Therefore it starts to execute instructions at address $000000.

With J25 in position 1, low memory is supposed to be dynamic RAM, but because there isn't any, IC36 gets no data acknowledge, so the 68000 quits with a bus error almost immediately after reset. With J25 in position 2, however, the EPROM is supposed to be mapped into low memory. There isn't any EPROM, of course, but the address decoder and the $\overline{\text{DTACK}}$ generator don't know that; hence they generate $\overline{\text{DTACK}}$ as if the EPROM were there. The 68000, therefore, executes the OR program until it passes the highest EPROM address, after which $\overline{\text{DTACK}}$ disappears, and the system halts on a bus error.

Step 12: ROM and RAM

The circuitry for the EPROM and the static RAM was shown in Fig. 3 of the December installment. The 68000's data bus is 16 bits wide, but no one makes EPROM's or RAM's that have 16 data lines. The solution is to use two 8-bit-wide EPROM's and two 8-bit-wide static RAM's.

Three kinds of EPROM's can be used, but both EPROM's of a pair must be the same type. Two 27128's hold 32K bytes; two 27256's hold 64K bytes; and two 27512's holds 128K bytes. 27128's have enough capacity to hold our machine-language monitor (HUMBUG) and BASIC, but the prices on memory IC's are so unstable that kits may contain any of those, depending on current market conditions. Jumpers J19 and J20 are used to select the type of EPROM that is being used.

The static RAM's are pair of 6116's, which provide a total of 4K bytes of memory. But it is possible to replace one—or both—with a Mostek MK48T02, which has the same pinout as a 6116, but which also contains some RAM and two additional features: a clock/calendar (whose registers replace the top eight locations of the RAM), and a lithium battery that powers both the clock and the RAM when the computer is turned off. The battery is rated for at least 31,000 hours of operation, or slightly more than 3½ years. In actual use, it should last longer, because the battery powers the clock only when the computer is off. (Note that the MK48T02, if used, must be inserted into the IC28 socket, because the clock software in SK*DOS expects to find the IC in the lower byte.)

Integrated circuits IC20 and IC21 handle the upper eight bits of the data bus, and IC27 and IC28 handle the lower eight bits. That division may be confusing and hence needs explanation. When the 68000 stores a word (two bytes) from an internal register into memory, the left byte (also called the more significant byte, or the upper byte) goes into memory first, and is always in an even-numbered location. The right byte (also called the least significant byte, or the lower byte) is stored in the next available memory location, an odd number. For example, the number $1234 might be stored as $12 in location $3500 and $34 in location $3501. That arrangement is logical, but it is confusing for two reasons: (1) the upper byte is actually stored in the lower address, and (2) other microprocessors (particularly those in the Intel family) store words in the opposite order. $1234 would be stored in sequential locations as $34 followed by $12.

Further complicating things is the fact that both EPROM's are controlled by the same $\overline{\text{CE}}$ (chip enable) and $\overline{\text{OE}}$ (output enable) signals, whereas there are separate enable signals for each of the static RAM's. Reading both EPROM's simultaneously does no harm; if the 68000 only wants one byte, it simply ignores the other half of the data bus. But writing to the static RAM requires two control lines to make sure that writing to one RAM does not inadvertently store unwanted data in the other.

Note also the difference between $\overline{\text{CE}}$ and $\overline{\text{OE}}$—when RAM or EPROM is read, $\overline{\text{CE}}$ enables the IC and starts the read process, but data does not appear on the bus until $\overline{\text{OE}}$ is asserted. In many systems, $\overline{\text{CE}}$ is used to put the entire chip into a low-power mode when it is not being accessed. In the PT-68K we control both lines together so the IC switches into low-power mode after every access.

Earlier we said that there is no A0 on the address bus; now let's see how the system gets by without it. Consider the static RAM, for example, which starts at $FF0000. Memory locations are located in specific IC's as follows. All the even addresses are stored sequentially in IC21 ($FF0000 is in location 0, $FF0002 is in location 1, $FF0004 is in location 2, etc.), and all the odd addresses are stored sequentially in IC28 ($FF0001 is in location 0, $FF0003 is in location 1, etc.).

The address lines are shifted by one bit because shifting a binary number to the right by one bit divides the number by two. For example, location $FF0008 ends with the bits 1000; physically, that byte would be located in address 0100 (i.e., address 4) of IC21. Location $FF0009, on the other hand, ends with the bits 1001; it would also be in location 0100, but in IC28. So the last bit of an address tells us which IC it is stored in.

Now that we understand how the EPROM and static RAM circuitry works, let's connect it. Remove the Molex pins from the IC21 and IC27 sockets, and install the following components: a socket for IC26 (74LS32), C12 and C66 (0.1 μF), sockets for IC21 and IC28 (EPROM's), and three-pin header strips at J19 and J20.

Then install the EPROM marked *Upper* in IC20 and the EPROM marked *Lower* in IC27, and then the two 6116 static RAM's in IC21 and IC28. Before turning on the power, make sure that the jumpers are positioned correctly, according to the chart shown in Fig. 3 last time. Also, place jumper J18 in position 2 to address the EPROM at location $000000. Now turn on the power.

If all is well, the HALT LED should go off after about a second. If it does, that's a pretty good sign that there are no serious problems, even though the computer is still not fully operational. If it doesn't, recheck all connections and parts installed since last time.

For more confidence, use the LED probe to check a few

continued on page 102

FLOPPY-DISK DATA STORAGE

continued from page 94

Breaking copy protection

Copy protection—both making it and breaking it—is big business. There are two fundamental ways to get around copy protection. The first is to buy a program that knows how to copy protected disks. If you're lucky, it will know how to make sense of the particular protection scheme(s) used on your disk and will be able to copy it with no muss and no fuss. But the price you pay for that kind of mindless copying is that, if the program can't copy the disk, there isn't a thing you can do except try some other copy program.

The second method is to use some disk tools to snoop through the disk and remove the copy protection yourself. As with most things, each method has advantages and disadvantages. The method you choose depends on how good your tools are, how well you can use them, how much you know about your computer, how much time you want to spend, and how badly you want to make the copy.

There's simply not space to go into the details of how to break copy protection; it's an art in itself. Basically, it involves spending endless hours at a totally unnatural act: staring at and trying to decipher page after page of undocumented object code. Unless you've actually done it, there just aren't any words to describe the amount of work involved.

One reason is that the code you'll be looking at probably was written to be confusing. The code in Listing 2 is a perfect example. Go through it and see whether you can understand what's going on. It's real code from a popular Apple game. There are no tricks here, either. It's just that things aren't what they seem to be. The real meaning (and the real code) is hidden. If you figure it out, drop us a note. And if there's enough response, we'll take up the subject of copy breaking in another article.

If you want to learn about the in's and out's of copy protection, you'll have to get familiar with the normal workings of the standard DOS used by your computer. Read the books mentioned in the References sidebar, and follow their tutorials. Those books won't tell you how to break copy protection, but they'll give you the basic tools you need to do so.

If you just want to back up your copy-protected (IBM) software, try the Option Board from Central Point Software. It's the ultimate tool for dealing with disks on a bits-and-bytes level, and it also helps copy "un-copyable" software. All the screen dumps in this article were produced using the Option Board, which also reads disks formatted on just about any computer, including IBM, CP/M, and even the Apple! That's a major accomplishment—especially for a $100 piece of hardware.

You'll also find a list of some good copy programs in the References sidebar; you should have some of them in your library even if you don't need to copy protected disks. Whenever you get a floppy-disk data error, chances are what has happened is that at least one sector was written incorrectly. All it takes is one bad bit in a header and the sector will be unreadable by DOS. If the damage is in the directory, you'll be unable to access any of the data and will be faced with the thankless job of trying to reconstruct your files. The point is that many programs capable of dealing with protected disks can deal with damaged disks as well.

BUILD THE PT-68K

continued from page 101

signals. First, all data-bus lines should have pulses. Likewise, address lines A1–A18 should have lots of pulses, but A19–A23 should not dim the LED at all. If HUMBUG (the EPROM monitor) is running, it is accessing mostly EPROM locations $F80000 and above, and static RAM locations $FF0000 and above. Since the first five bits of both $F8 and $FF are 11111, the LED probe should indicate a high when it is connected to each line.

You can also test the outputs of the address decoder. You should note many pulses on pin 19 of IC63, which decodes all addresses above $F80000. You should also see pulses on IC64, pin 4 (which selects the EPROM), pin 7 (which drives IC64-b), and pin 10 (which selects the static RAM.)

What you can't see on the LED (although you can on a good scope) are very brief pulses on the high-order address lines. The 68000 is trying to access error vectors (pointers to special software routines) because it senses a bus error due to the fact that no DUART (serial port) is installed. Likewise, you can see brief pulses on IC64, pin 5, as HUMBUG is desperately trying to access the expansion slots in the hope that there is a video board there to report the error on.

Step 13: Running at last!

Now we can finally get the computer to do something useful. Install IC10 (MC68681 DUART) and its socket, 10K resistors R9, R10, R12, R13, and IC3 (3.6864 oscillator) on the board. The pointed corner of the oscillator is pin 1. Last, connect a speaker to J18. The small speaker that comes with PC clone cabinets is best; any other inexpensive speaker will do. Just don't connect one of your stereo system's speakers!

Now turn on the power; about a second later, the HALT LED should go off, and another second later you should hear a beep-boop from the speaker. That's a signal from HUMBUG that it is running, and it's the first indication that there are no serious problems—even the slightest problem occurring would prevent HUMBUG from sounding that dual tone.

Now that a real program is running with no errors, the signals in the address decoder will be slightly different. HUMBUG is currently running a loop that continuously checks for the presence of a keyboard; that loop runs entirely in EPROM and calls the first DUART, so you will see pulses on the $\overline{\text{EPROM}}$ chip-select line, IC64-a, pin 4; the $\overline{\text{I/O}}$ line on IC64-b, pin 12; and the $\overline{\text{I/O0}}$ line, IC34, pin 15. You will no longer see pulses on the $\overline{\text{SRAM}}$ chip select line.

Next time we will continue building the I/O section of the PT-68K so we can communicate with it. See you then.